

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A261 367



DTIC
ELECTE
S MAR 9 1993 D
C

**Research Directions in Software Analysis,
Synthesis and Certification**

Luqi

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Monterey, California 93943

93 3 3 000

93-04964



NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R.W. West, Jr.
Superintendent

HARRISON SHULL
Provost

This report was prepared with research funded by the Naval Research Funds provided by the Naval Postgraduate School.

Reproduction of all or part of this report is authorized.

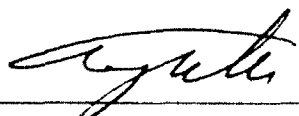
This report was prepared by:

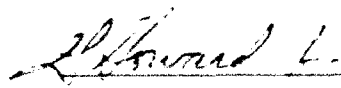


LUQI, Associate Professor

Reviewed by:

Released by:



THOMAS WU
Associate Chairman for
Technical Research

PAUL MARTO
Dean of Research

REPORT DOCUMENTATION PAGE

| | | | | | |
|---|-------|--|--|---|----------------------------------|
| 1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED | | | 1b RESTRICTIVE MARKINGS | | |
| 2a SECURITY CLASSIFICATION AUTHORITY | | | 3 DISTRIBUTION AVAILABILITY OF REPORT Approved for public release, distribution is unlimited | | |
| 2b DECLASSIFICATION/DOWNGRADING SCHEDULE | | | | | |
| 4 PERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-92-018 | | | 5 MONITORING ORGANIZATION REPORT NUMBER(S) | | |
| 6a NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School | | 6b OFFICE SYMBOL (if applicable) CS | 7a NAME OF MONITORING ORGANIZATION | | |
| 6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943 | | | 7b ADDRESS (City, State, and ZIP Code) | | |
| 8a NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School | | 8b OFFICE SYMBOL (if applicable) | 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER | | |
| 8c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943 | | | 10 SOURCE OF FUNDING NUMBERS | | |
| | | | PROGRAM ELEMENT NO | PROJECT NO | TASK NO |
| | | | WORK UNIT ACCESSION NO | | |
| 11 TITLE (Include Security Classification) Research Directions in Software Analysis, Synthesis and Certification | | | | | |
| 12 PERSONAL AUTHOR(S) Luqi | | | | | |
| 13a TYPE OF REPORT | | 13b TIME COVERED FROM _____ TO _____ | | 14 DATE OF REPORT (Year, Month, Day) 1992, December, 31 | |
| | | | | 15 PAGE COUNT 13 | |
| 16 SUPPLEMENTARY NOTATION | | | | | |
| 17 COSATI CODES | | | 18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) | | |
| FIELD | GROUP | SUB-GROUP | | | |
| | | | | | |
| | | | | | |
| 19 ABSTRACT (Continue on reverse if necessary and identify by block number) This paper presents a view of research directions relevant to producing reliable and useful software systems. Appropriate research goals are identified for achieving improvements in software quality via formalization and computer aid for software analysis, synthesis, and certification tasks at all states of software development and evolution. | | | | | |
| 20 DISTRIBUTION AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS | | | 21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED | | |
| 22a NAME OF RESPONSIBLE INDIVIDUAL Luqi | | | 22b TELEPHONE (Include Area Code) (408) 656-2735 | | 22c OFFICE SYMBOL CSLq |

Research Directions in Software Analysis, Synthesis and Certification

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper presents a view of research directions relevant to producing reliable and useful software systems. Appropriate research goals are identified for achieving improvements in software quality via formalization and computer aid for software analysis, synthesis, and certification tasks at all stages of software development and evolution.

1. Introduction

Systematically producing and adapting reliable software that meets the needs of its clients is a dominant software engineering problem in the 1990's. Reliability of software products is a concern particularly for systems whose malfunction may result in lost lives, injuries, or financial losses. It is practically impossible to produce error-free software systems that solve real (complex) problems by purely manual development methods because human error rates are too high. Sound automatable methods for software analysis, synthesis and certification are needed to bridge this gap. For practical impact, these methods must fit together to cover the entire software development and evolution process. Software evolution (sometimes called maintenance) is a major concern in this context because errors are often introduced as a system is modified, and evolution typically accounts for more than half of the effort of developing and supporting a software system.

Reliability concerns indicate that we need automated assistance in all stages of software development, but complete automation of software development and evolution is not feasible in the near future. Some realistic research goals in this situation include:

- (1) Formulating a consistent set of accurate mathematical models for a set of subproblems covering the software development process. This is needed to enable integration of the methods and tools for solving problems related to different aspects of software development.
- (2) Developing and certifying the correctness of completely automated synthesis methods for the tractable subproblems. In cases where this is possible, this approach provides both reliability and productivity gains.
- (3) Developing interactive synthesis methods that guarantee absence of errors for less tractable subproblems. This approach combines the benefits of human creativity with the accuracy provided by computer application of sound formal methods.
- (4) Improving analysis and certification methods capable of detecting and diagnosing errors for subproblems that cannot be covered by error prevention techniques. If aspects of the process must remain manual, then automated assistance for locating and removing errors and for certifying that no errors are left are needed for those

| | |
|--------------------|-------------------------------------|
| SEARCHED FOR | |
| INDEXED | <input checked="" type="checkbox"/> |
| FILED | <input checked="" type="checkbox"/> |
| ABSTRACTED | <input checked="" type="checkbox"/> |
| REPRODUCED | <input type="checkbox"/> |
| By | |
| On | |
| Availability Codes | |
| Available for | |
| Special | |

A-1

aspects.

Most of the past work on improving software reliability has followed the last approach. This direction has enabled researchers to provide some useful tools without waiting for full understanding of all of the problems involved. This direction is also the least desirable one for the future because the search for errors is often very labor-intensive and because it is difficult to predict how many iterations will be needed to eliminate all of the errors. Thus error detection work is most reasonable for those aspects of software development for which error prevention techniques are not feasible.

2. Assessing the State of the Art

Past approaches have considered isolated subproblems of the real software reliability problem, mostly at the code level.

Work on software testing has produced a few solid results in addition to many heuristics whose effectiveness is difficult to evaluate. Successful execution of test sets constructed by random sampling from a probability distribution can provide lower bounds on the mean number of executions between failures if actual input values correspond to the given probability distribution [6]. This kind of statistical reliability assurance is sufficient in cases where input distributions are predictable and non-zero failure rates can be tolerated. Statistical assurances are not sufficient for critical applications where even one failure is unacceptable. Statistical reliability measures can also be misleading if real input distributions are unstable or unknown, because there exist input distributions with high failure rates for any deterministic program that is not completely error-free.

For some specialized classes of programs, there exist methods for constructing a finite set of test cases whose successful execution can establish correctness of the program for all possible inputs [4,6]. This is not possible in the general case: testing can show the presence of software errors but it cannot certify their absence for unconstrained programs.

Work on program verification has sought to construct and mechanically check mathematical proofs that given programs meet given specifications for all possible inputs. Methods for doing this exist, but this technology is not yet mature for practical applications, particularly with respect to tool support and range of applicability of the formal models. Weaknesses of current technology include the following.

- (1) Proving that a program satisfies a given specification is useless without some assurance that the specification is *valid*, i.e. that it accurately represents the needs of the users of the program. Validation of specifications is a major task and systematic methods for doing this are not well developed.
- (2) It is impossible to prove that an incorrect program satisfies its specification. Past studies indicate that it takes much more effort to construct code that fully meets a specification than it does to mathematically verify a correct program [5].
- (3) Current systems require considerable human assistance, and the mathematical skills required to use these systems are beyond the abilities of many practicing software developers.

3. Future Opportunities

More recently proposed approaches focus on *error prevention* rather than error detection. Error prevention is possible both in cases where a software development task can be completely automated, and in cases where an automated tool realizes all of the designer's decisions in constrained ways that do not allow the designer to make a mistake, or that eliminate some kinds of mistakes. Some examples are meaning-preserving software transformations, which prevent divergences between specifications and the code [2], and syntax-directed editors, which prevent the creation of programs that do not conform to the syntax of the programming languages.

It is commonly believed that error prevention is more difficult than error detection, but this is not always the case. For example, checking whether an equational specification for an abstract data type is consistent and complete is an undecidable problem. Nevertheless, there exists an error prevention technique that guarantees that every specification that can be generated according to the rules is complete and consistent. These rules are simple enough to be applied and checked by a text editor, and they are sufficiently loose to accommodate the styles of specification that normally occur in practice [1].

Software development deals with information of many different kinds, at different levels of abstraction. We summarize some of the types of software analysis, synthesis and certification problems that should be investigated for several levels of software representations in Fig. 1.

3.1. Requirements Level

The requirements level establishes the goals for a proposed system and formulates models of the problem and the expected environment of the proposed system.

An important aspect of requirements analysis is achieving and maintaining consistency as the analysts discover and record the requirements. A promising approach to this problem is providing automated support for calculating and maintaining derived properties and consequences of the requirements, and for tracing dependencies to determine the causes of conflicts and inconsistencies. Better algorithms for this process and primitives suitable for expressing and effectively maintaining dependencies in software requirements should be investigated.

Another aspect of requirements analysis is modeling the environment of a proposed system. Especially for embedded software systems, an accurate formal characterization of the system to be controlled is essential for assessing the effectiveness of the control software. The environment of such a system must often be simulated or otherwise formally analyzed to enable safe and meaningful testing or analysis of the embedded software system. Systematic methods for validating and testing the formal models of the environment against the properties of the actual physical systems they represent are needed. Both analytical and experimental methods should be explored to establish that the formal environment models used in other software analysis and testing activities are adequate representations of reality.

Many critical software systems are embedded systems, which means that the software is part of a larger system. Thus an essential part of checking the adequacy of

| Level | Type of Analysis/Synthesis |
|---------------|---|
| Requirements | consistency: truth maintenance model validation: simulation and proof subgoal verification: prototyping and proof |
| Specification | adequacy: prototyping, operational scenarios consistency: type and domain checking safety: proofs validation: paraphrasing, views, simplification |
| Design | verification: proof of decomposition liveness: deadlock and starvation checking robustness: impact of degraded hardware design for testing: control and observation performance: complexity analysis feasibility: satisfiability proofs |
| Coding | synthesis: meaning-preserving transformations performance: time and space analysis liveness: proof of (clean) termination real-time: analysis of scheduling methods generic units: analysis of component families error detection: complete test sets error location: weakest preconditions |
| Evolution | change impact: symbolic differences restructuring: meaning-preserving transformations error prevention: change merging |

Fig. 1 Types of Software Analysis and Testing

the requirements for the software is checking that any system meeting the requirements will be sufficient to meet the requirements for the larger system in its intended operational context. Hard real-time constraints in an embedded system are often motivated by the requirement to control the larger system to ensure it remains within a given range of operating states. For example, the cycle rate of an auto-pilot must be sufficiently high to ensure that the airplane remains within a given radius of its planned position at all times. At the current time, lower level requirements are usually formulated based on past experience and informal guidelines rather than on systematic derivations or verification procedures with respect to the higher level requirements.

Both formal and experimental methods for systematically establishing such properties are needed. Required supporting technology for this process includes computer-aided construction of prototypes [7].

3.2. Specification Level

The specification level is concerned with defining the interface of a proposed system, both at the functional and the command representation levels.

The primary measure of the adequacy of a specified interface is whether it will meet the needs of the user. This question is best addressed by experimental rather than analytical techniques because it addresses the problem of checking the correspondence between a formalized specification and the actual and informal needs of the users. One way of approaching this problem is via prototyping and operational scenarios. Operational scenarios are common tasks in the customer's problem domain, expressed in the user's terms. Such scenarios serve as test cases for the specifications, whose purpose is to determine whether a proposed interface is adequate for carrying out all of the tasks the users will have to perform. Such a test passes if the facilities provided by the proposed system interface can be combined to carry out the tasks in the operational scenario, and provide a systematic means for exercising a prototype in a demonstration to the users. Systematic methods for deriving sets of scenarios from a requirements document, coverage criteria, and experimental evaluation of the effects of such coverage criteria on change requests to the affected interfaces during system maintenance should be investigated.

A related concern is validating a formal specification, to ensure that it correctly captures the intentions of the users. While this is an informal process, it can be aided by formalized and automatable procedures. Symbolic execution and operational scenarios can be applied to this problem. Proof techniques can also be used in this way [8], by obtaining a list of expected system properties and then proving that the formal specification has those properties. Other relevant processes involved are paraphrasing, projection, and simplification. Paraphrasing is the process of transforming a formal specification into a form that a user can understand, while preserving its meaning. Projection is the process of extracting the parts of a specification relevant to a particular user or task, while hiding other details. Simplification is the process of transforming a formal specification into a simpler form with an equivalent meaning. These three processes can be combined to help users selectively review formal specifications using representations they can understand. The research questions in this area concern certifying the transformations to ensure they preserve the meaning of the specification and experimentally evaluating the effectiveness of different representations for communicating with untrained users.

Consistency of a specification is another common concern, especially for large and complex systems. Since consistency is a property of a formal document, it can be addressed by analytical techniques. Some aspects of consistency checking that need further development are type and domain consistency checking. Type checking at the specification level is more difficult than the corresponding problem at the code level because types can have subtypes defined by semantic considerations. Domain

checking is the process of ensuring that partial functions or predicates are used only within their domain of definition, and that partially defined generic units are instantiated only with actual parameters in their respective domains of definition. Logical inference capabilities are necessary for both of these kinds of specification analysis.

Another concern with formal specifications is checking safety properties. For example, past research projects have been concerned with whether a proposed operating systems kernel satisfied certain security properties, such as the impossibility of transmitting classified information from a process with a high security classification to an unauthorized process. The goals of safety analysis procedures are to identify cases where the specifications allow behaviors violating the safety properties, or to certify that no such cases exist. Systematic procedures for this process are needed because the connection between a formal specification and a safety property can be quite indirect and can require extensive reasoning and analysis to establish.

3.3. Design Level

The design level is concerned with the decomposition of a problem into a hierarchical structure of independent modules. Such a decomposition consists of interconnection information and formal specifications for the components.

The primary reliability property of a decomposition structure is whether it will correctly realize the specification at the next higher level. This problem is subject to mathematical proof techniques. The problem is easier to solve than the general proof of correctness problem at the code level because the module interconnection language is can be considerably simpler than a programming language. Most of the analysis can be carried out at the specification level, since the problem is to check whether a given combination of specified components will satisfy the required properties of the composite. Research questions in this area involve the best choice of interconnection primitives to support effective and efficient inference procedures, and formal characterizations of interfaces and resource models sufficient to ensure correctness of decompositions in the presence of distributed processing and hard real-time constraints.

Another type of property of interest for parallel and distributed systems is liveness. Techniques for checking for potential deadlock or starvation conditions in such a design are desired. Deadlock detection appears to be decidable for asynchronous models and undecidable for synchronous ones. Some research problems in this area include developing solutions to the problem in the error prevention style, and developing better symbolic techniques for deadlock detection and diagnosis. The problem with past finite-state techniques for deadlock detection is that practical applications often have state spaces that are much too large for enumeration techniques to be applied in practice. Formal approaches that can avoid state enumeration via induction and logical reasoning should be developed further.

An important class of analysis involves the effects of degraded hardware on the properties of a design, relative to a mapping of software components to hardware components. This kind of analysis is essential for achieving reliable fault tolerant systems, especially those with distributed implementations. In addition to certifying that proposed configurations realize given degrees of fault tolerance, automatic derivation of

the implied constraints on allocation of software functions to hardware units is desirable.

Evaluation of a design for time and space performance is another kind of software analysis that has potential importance. Automated support for classical complexity analysis is needed, along with estimates for the ranges of input sizes and constant factors determined by classes of algorithms.

A final consideration is satisfiability. The satisfiability of a specification can be established if an implementation can be produced and certified to be correct. However, it would be useful to determine whether it is possible to satisfy a given specification before the implementation is attempted, and in cases where it is not, to characterize the set of inputs for which the specification is impossible to meet. Analytical techniques for constructing weakest infeasible preconditions characterizing this set of inputs should be explored. Since it is possible to derive programs from sufficiently constructive proofs of the satisfiability of a specification, further research along these lines has good potential for producing a practical impact.

3.4. Code Level

The best way to achieve quality is to systematically prevent errors. Automatable methods for synthesis of efficient code from formal specifications via meaning-preserving transformations should be investigated. Of particular interest are systems that can choose transformations without explicit human guidance, or with guidance from general declarative advice that can be formulated without explicit reference to the details of the current state of the derivation and does not require explicit human interaction during the derivation process.

Accurate performance analysis requires detailed code and knowledge about properties of a particular compiler and target machine. Generic table driven methods for performing such analysis, and for relating design-level properties of abstract algorithms to detailed properties of actual machine-level implementations and compiler optimizations is needed to accurately certify correctness of programs with hard real-time and real-space constraints. Research problems in this area include formal modeling of implementation-specific properties and constraints in ways that can be combined with implementation-independent analyses of abstract programs.

Another problem is certification of clean termination. This problem gains new dimensions in parallel and distributed systems, where termination can be influenced by scheduling properties and hardware failures. Research questions include models and techniques for analyzing programs in these domains.

Analysis of real-time systems includes analysis of scheduling methods to determine whether a proposed scheduling discipline will meet specified deadlines under all possible operating conditions. Research questions in this area have flexible scheduling methods, the effects of shared resources, overload resolution policies, and remote communications as major concerns.

The problem of certifying generic code units or families of related programs generated by meta-programming schemes is a major concern in systems for managing reusable software. A software component is most effectively re-used if it is flexible

and can be adapted to many needs. Such a component often corresponds to a family of related program units with an unbounded number of elements. The problems of testing and analyzing the reliability of such program families is an important research question.

Classical testing approaches need more foundational work on the construction of complete test sets. A complete test set is a set of test cases which is guaranteed to detect any error in a particular well-defined class of errors. More work is needed on the construction of finite complete test sets, and on characterizing the set of faults whose absence is guaranteed by successful execution of the test set. Such work should include automated techniques for constructing the required test oracles from the formal specifications of the code to be tested. A weakness of statistical approaches to testing is the size of the test set required for certifying that systems have low failure frequencies, which makes manual examination of test results impractical. To apply these techniques in practice, we need automated methods for deciding whether or not the outputs produced by a test case conform to a specification.

An aspect of code analysis of great practical importance is error location. One approach to this problem is to derive weakest preconditions for suspected pieces of code, to characterize the space of inputs for which the code fails. This and other approaches should be refined and evaluated to determine their practical utility.

3.5. Evolution Aspects

Software maintenance is acknowledged to be more difficult and error prone than the initial development. An important kind of software analysis for this part of software development is characterization of the effects of a change to a software system. Symbolic representations for the parts of the input space and the output space of a program affected by a given change to the code are useful for testing and evaluating a modification for conformance with the expected results. Computer-aided identification of the parts of a specification affected by a given requirements change, the parts of a design affected by a given specification change, and the parts of the code affected by a given design change are also important areas for research.

When changing a software system, it is often necessary to reverse an earlier design decision while preserving the later ones, and to combine the effects of several changes that were developed independently. Both of these problems can be addressed by change-merging techniques [3]. Automated construction and application of monotonic transformations that can modify specifications and programs in response to problems with the current behavior of a system is also an important research problem.

4. Conclusion

Advances in software analysis, synthesis and certification are essential for realizing trusted software systems. Work in this area should be expanded beyond the traditional domains of testing code in a programming language and proving that programs satisfy formal specifications, to include software products at all stages of development from requirements analysis to system evolution. Some key areas for future research are

- (1) Methods for validating requirements and specifications, such as prototyping and techniques for testing prototypes and specifications relative to user perceptions,
- (2) Methods for constructing programs that guarantee correctness with respect to formal specifications, such as program synthesis by meaning-preserving transformations and certification of application-specific program generation schemes.
- (3) Approaches for making formal methods easier to use, reducing the amount of manual effort required, and for reducing the amount of training and mathematical skill required for practitioners to apply these methods by designing software tools that hide theoretical complexities and provide simple interfaces.
- (4) Methods relevant to software evolution, such as change merging, monotonic transformations for modifying specifications and programs, and incremental versions of conventional software analysis, synthesis, and certification methods.
- (5) Software analysis techniques addressing properties of parallel, distributed, real-time, and knowledge-based systems should be explored as well as those for sequential systems.
- (6) Further work on program testing is needed, to expand the domains in which firm conclusions about satisfying specifications can be drawn from finite sets of test cases constructed by definite and effective methods, and to systematically check assumptions about the operating environment on which the design of a software system depends.

1. S. Antoy, P. Forcheri and M. Molino, "Specification-based Code Generation", in *Proc. 23rd Hawaii International Conference on System Sciences*, IEEE Computer Society, Jan. 1990, 165-173.
2. F. Bauer, B. Moller, H. Partsch and P. Pepper, "Formal Program Construction by Transformations - Computer-Aided, Intuition-Guided Programming", *IEEE Trans. on Software Eng.* 15, 2 (Feb. 1989), 165-180.
3. V. Berzins, "Software Merge: Models and Methods", *International Journal on Systems Integration* 1, 2 (Aug. 1991), 121-141.
4. J. Bicevskis, J. Borozovs, U. Straujums, A. Zarins and E. Miller, Jr., "SMOTL - A System to Construct Samples for Data Processing Program Debugging", *IEEE Trans. on Software Eng.* SE-5, 1 (Jan. 1979), 60-66.
5. D. Good, "Mechanical Proofs about Computer Programs", Technical Report #41, Institute for Computing Science, University of Texas at Austin, Mar. 1981.
6. W. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, New York, 1987.
7. Luqi, "Software Evolution via Rapid Prototyping", *IEEE Computer* 22, 5 (May 1989), 13-25.
8. A. Mili, W. Xiao-Yang and Y. Qing, "Specification Methodology: An Integrated Relational Approach", *Software Practice and Experience* 16, 11 (Nov. 1986), 1003-1030.

INITIAL DISTRIBUTION LIST

| | |
|---|----|
| Defense Technical Information Center Cameron Station Alexandria, VA 22314 | 2 |
| Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5100 | 2 |
| Office of Research Administration Code 08 Naval Postgraduate School Monterey, CA 93943-5100 | 1 |
| Chief of Naval Research 800 N. Quincy Street Arlington, VA 22302-0268 | 1 |
| Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268 | 1 |
| Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000 | 1 |
| Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100 | 2 |
| Dr. Luqi Computer Science Department, Code CSLq Naval Postgraduate School Monterey, CA 93943-5100 | 10 |
| Dr. Thomas Wu Computer Science Department, Code CSWq Naval Postgraduate School Monterey, CA 93943-5100 | 1 |